University of Arizona, Department of Computer Science

CSc 340 Foundations of Computer Systems — Assignment 2

Christian Collberg

Due Thursday, February 22, at 11:59pm

1 Assignment

Many programs these days are written in *interpretive languages*. This means that a compiler for the language does not produce native machine code (such as MIPS-code, Sparc-code, etc.) but rather code for a *virtual machine* (VM). Java, for example, is compiled to *Java bytecode* which is executed by a Java VM.

In this assignment we are going to implement an interpreter for **NQJBC**[1], essentially a subset of Java bytecode. Since we want our interpreter to be as fast as possible, we decide to implement it in assembly-code.

The file `/home/cs340/spring01/prog2/interp.java` contains a simple implementation of the interpreter in Java. The file `/home/cs340/spring01/prog2/interpret.s` contains a skeleton program to get you started writing the MIPS assembly-code version. Much has already been implemented for you: a test harness, and the **NQJBC** instructions `PUSHB`, `PRINT`, `PRINTLN`, and `EXIT`. You will implement the remaining bytecodes.

2 The NQJBC Bytecode

Table1 shows the instructions available in **NQJBC** and their semantics.

Like Java bytecode, **NQJBC** is a *stack-machine*. This means that all communication between instructions in a **NQJBC** program takes place on an evaluation stack, rather than through registers, as on the MIPS. In our implementation the stack consists of 100 integers and a stack-pointer, the `SP`.

Here is a more detailed description of the **NQJBC** instruction set:

⌜`ADD`⌝: Pop the two top integers A and B off the stack, then push $A + B$.

⌜`SUB`⌝: As above, but push $A - B$.

⌜`MUL`⌝: As above, but push $A * B$.

⌜`DIV`⌝: As above, but push A/B.

⌜`PUSHB` X⌝: Push X, a signed, byte-size, value, on the stack.

⌜`PRINT`⌝: Pop the top integer off the stack and print it.

⌜`PRINTLN`⌝: Print a newline character.

⌜`EXIT`⌝: Exit the interpreter.

[1] Not Quite Java ByteCode™. Not to be confused with **NKOTB**.

mnemonic	opcode	stack-pre	stack-post	side-effects
ADD	0	[A,B]	[A+B]	
SUB	1	[A,B]	[A-B]	
MUL	2	[A,B]	[A*B]	
DIV	3	[A,B]	[A-B]	
PUSHB X	6	[]	[X]	
PRINT	7	[A]	[]	Print A
PRINTLN	8	[]	[]	Print a newline
EXIT	9	[]	[]	The interpreter exits
BEQ L	12	[A,B]	[]	if A=B then PC+=L
BNE L	13	[A,B]	[]	if A!=B then PC+=L
BLT L	14	[A,B]	[]	if A<B then PC+=L
BGT L	15	[A,B]	[]	if A>B then PC+=L
BLE L	16	[A,B]	[]	if A<=B then PC+=L
BGE L	17	[A,B]	[]	if A>=B then PC+=L
BRA L	18	[]	[]	PC+=L

Table 1: **NQJBC** bytecode semantics.

⌜`BEQ` L⌝: Pop the two top integers A and B off the stack, if $A == B$ then continue with instruction $\texttt{PC} + L$, where `PC` is address of the instruction *following* this one. Otherwise, continue with the next instruction.

⌜`BNE` L⌝: As above, but branch if $A \neq B$.

⌜`BLT` L⌝: As above, but branch if $A < B$.

⌜`BGT` L⌝: As above, but branch if $A > B$.

⌜`BLE` L⌝: As above, but branch if $A \leq B$.

⌜`BGE` L⌝: As above, but branch if $A \geq B$.

⌜`BRA` L⌝: Continue with instruction $\texttt{PC} + L$, where `PC` is address of the instruction *following* this one.

3 Examples

Both `interp.java` and `interpret.java` contain a number of example programs written in **NQJBC**.

This program prints a newline character and then exits:

```
PRINTLN
EXIT
```

This program prints the number 10, then a newline character, and then exits:

```
PUSHB 10
PRINT
PRINTLN
EXIT
```

This program pushes two values on the stack, then performs an `ADD` instruction which pops these two values off the stack, adds them, and pushes the result. `PRINT` then pops this value off the stack and prints it:

```
PUSHB 10
PUSHB 20
ADD
PRINT
PRINTLN
EXIT
```

This program computes 10+20*40+50 and prints the result:

```
PUSHB 10
PUSHB 20
PUSHB 40
MUL
ADD
PUSHB 50
ADD
PRINT
PRINTLN
EXIT
```

The skeleton file also contains several test routines that call the interpreter on the **NQJBC** programs above. Here's the output when I run the program using `spim`'s "batch" mode:

```
> spim -file interpret.s

Executing "program0":


Executing "program1":
10

Executing "program2":
30

Executing "program3":
860

Executing "program4":
100

Executing "program5":
1287

Executing "program6":
1026
```

```
Executing "program7":
1
2
3
4
5
6
7
8
9
```

4 Honors section

Those enrolled in the honors section should do their work on `wonka.cs.arizona.edu`. You are allowed to collaborate in converting the skeleton file to *wonka-normal-form*.

You should also implement at least Extension 1, for which you will receive no extra credit.

5 Extension 1 [10% extra credit]

For extra credit extend **NQJBC** to have access to a small memory area, essentially an array of 256 integers, numbered 0–255. Implement the `LOAD` and `STORE` instructions which access these memory cells.

⌜`LOAD` X⌝ pushes the contents of memory cell number X on the stack. ⌜`STORE` X⌝ pops the top integer off the stack and stores this value in memory cell number X. ⌜`ALOAD`⌝ and ⌜`ASTORE`⌝ are similar, but take both their arguments from the stack.

Also implement the ⌜`PUSHW` X⌝ instruction, which pushes X, a signed, word-size, value, on the stack, and ⌜`SWAP`⌝ which exchanges the two top elements on the stack.

Here is the semantics of these instructions:

mnemonic	opcode	stack-pre	stack-post	side-effects
LOAD X	4	[]	[Memory[X]]	
STORE X	5	[A]	[]	Memory[X] = A
ALOAD	19	[X]	[Memory[X]]	
ASTORE	20	[A,X]	[]	Memory[X] = A
PUSHW X	11	[]	[X]	
SWAP	21	[A,B]	[B,A]	

This program uses the `LOAD` and `STORE` instructions to store a value in memory cell number 7:

```
PUSHB 10
STORE 7
PUSHB 10
LOAD 7
```

```
MUL
PRINT
PRINTLN
EXIT
```

This program does the same thing, but using `ALOAD` and `ASTORE`:

```
PUSHB 10
PUSHB 7
ASTORE
PUSHB 10
PUSHB 7
ALOAD
MUL
PRINT
PRINTLN
EXIT
```

The final program is more interesting. It uses the branch instructions to construct a while-loop that prints out the numbers 1 through 9. The current value to be printed is in memory cell 1.

```
PUSHB 1         // mem[1] = 1;
STORE 1         //
LOAD 1          // if mem[1] < 10 goto exit
PUSHB 10        //
BGE 13          //
LOAD 1          // print mem[i] value
PRINT           //
PRINTLN         //
PUSHB 1         // mem[1]++
LOAD 1          //
ADD             //
STORE 1         //
BRA -19         // goto top of loop
EXIT            //
```

6 Extension 2 [10% extra credit]

Write a **NQJBC** program that implements your favorite sort routine, such as SelectionSort or BubbleSort. The program should store the integer sequence $\langle 7, 34, 99, 1, 23, 56, 11, 9, 87, 22 \rangle$ in memory cells 0–9, sort them, and then print out the resulting sequence.

7 Extension 3 [10% extra credit]

Consider the implementation of a typical binary instruction like `ADD`:

```
case ADD    : {
   stack[sp-2]=stack[sp-2]+stack[sp-1]; sp--;
   pc++; break;
}
```

In this code-fragment there are three accesses to the stack: we load the values of the two top elements, then write the result back to the new stack top.

On modern machines memory accesses are *very* expensive, they can take 100 times longer (or more!) than accessing a register. In this extension you should therefore try to reduce the number of accesses to the stack.

One technique that is frequently used is to store the topmost stack element (`TOS`, *top-of-stack*) in a dedicated register. For example, after the instruction sequence `PUSH 10`, `PUSH 20`, `PUSH 30`, the stack would normally contain $\langle 10, 20, 30\rangle$. In an implementation that uses the `TOS`-optimization, however, the stack would contain $\langle 10, 20\rangle$ and `TOS=30`.

In our new implementation the `ADD`-instruction would look like this:

```
case ADD    : {
   stack[sp-2]=stack[sp-2]+TOS; sp--;
   pc++; break;
}
```

`TOS` should be allocated to a particular register which the interpreter engine uses for no other purpose.

8 Extension 4 [10% extra credit]

In the main interpreter loop you will find the following code segment:

```
lb      $t0,($s0)       # load next bytecode
beq     $t0,0,ADD       # branch to appropriate handler
beq     $t0,1,SUB
beq     $t0,2,MUL
beq     $t0,3,DIV
beq     $t0,4,LOAD
beq     $t0,5,STORE
beq     $t0,6,PUSHB
beq     $t0,7,PRINT
beq     $t0,8,PRINTLN
beq     $t0,9,EXIT
beq     $t0,11,PUSHW
beq     $t0,12,BEQ
beq     $t0,13,BNE
beq     $t0,14,BLT
beq     $t0,15,BGT
beq     $t0,16,BLE
beq     $t0,17,BGE
beq     $t0,18,BRA
```

mnemonic	opcode	stack-pre	stack-post	side-effects
FADD	19	[A,B]	[A+B]	
FSUB	20	[A,B]	[A-B]	
FMUL	21	[A,B]	[A*B]	
FDIV	22	[A,B]	[A-B]	
PUSHF X	23	[]	[X]	
PRINTF	24	[A]	[]	Print A
FBEQ L	25	[A,B]	[]	if A=B then PC+=L
FBNE L	26	[A,B]	[]	if A!=B then PC+=L
FBLT L	27	[A,B]	[]	if A<B then PC+=L
FBGT L	28	[A,B]	[]	if A>B then PC+=L
FBLE L	29	[A,B]	[]	if A<=B then PC+=L
FBGE L	30	[A,B]	[]	if A>=B then PC+=L
ItoF L	31	[I]	[(float)I]	
FtoI L	32	[F]	[(int)F]	

Table 2: **NQJBC** bytecode semantics for floating-point instructions.

This is obviously very inefficient code. Whenever we execute a `BRA` instruction we have to perform 18 tests! Can you do better? Hint: you may find the MIPS instruction ⌜`jr` *reg*⌝ useful.

9 Extension 5 [10% extra credit]

Add support for floating-point operations to the **NQJBC** instruction set. Floating-point numbers are 4 bytes wide and can hence be saved on the stack and in **NQJBC**'s memory area. Figure 2 gives the semantics of these instructions.

Here is a more detailed description of the new **NQJBC** instructions:

⌜`FADD`⌝: Pop the two top floating-point numbers A and B off the stack, then push $A + B$.

⌜`FSUB`⌝: As above, but push $A - B$.

⌜`FMUL`⌝: As above, but push $A * B$.

⌜`FDIV`⌝: As above, but push A/B.

⌜`PUSHF` X⌝: Push X, a signed, word-size, floating-point value, on the stack.

⌜`PRINTF`⌝: Pop the top floating-point number off the stack and print it.

⌜`FBEQ` L⌝: Pop the two top floating-point numbers A and B off the stack, if $A == B$ then continue with instruction $\texttt{PC} + L$, where `PC` is address of the instruction *following* this one. Otherwise, continue with the next instruction.

⌜`FBNE` L⌝: As above, but branch if $A \neq B$.

⌜`FBLT` L⌝: As above, but branch if $A < B$.

⌜`FBGT` L⌝: As above, but branch if $A > B$.

⌜FBLE L⌝: As above, but branch if $A \leq B$.

⌜FBGE L⌝: As above, but branch if $A \geq B$.

⌜ItoF L⌝: Pop the top integer value off the stack, convert it to a floating-point number, and push this value.

⌜FtoI L⌝: Pop the top floating-point number off the stack, convert it to an integer by truncation, and push this value.

You should also provide a number of test programs that exercise the new instructions.

10 Turnin and the Makefile

A sample makefile for your use can be found in file `/home/cs340/spring01/prog2/Makefile`.

When you have completed the program, submit your program file by typing ⌜**`make turnin`**⌝. Or, to do it manually type ⌜**`turnin cs340prog2 interpret.s`**⌝. If you have implemented any of the extensions you should describe them in the README file, so that the graders know what to test for.

This is an individual assignment. Don't show your code to anyone, don't read anyone's code, don't discuss the details of the code with anyone. If you need help with the assignment see the TA or the instructor.

A Stack Machine Example

When the program ⌜`PUSHB 10, PUSHB 20, PUSHB 40, MUL, ADD, PUSHB 50, ADD, PRINT`⌝ is executed the following operations are performed on the stack: